Tarea 5:
Huffman Code Compression

Nombre: Franco Zautzik

Email: francozautzik@gmail.com

Profesor/a: Patricio Poblete

Auxiliares: Gabriel Flores

Sven Reisenegger

Fecha de entrega: 16/07/2018

Introducción:

Se plantea el problema de encontrar una manera de poder comprimir un archivo de texto logrando así menor utilización de memoria. El método de compresión de Huffman es un solución común a este problema donde, a partir de la frecuencia de apariciones de los caracteres en el texto, se genera una codificación para este en bits.

El programa que que se busca como solución debía, a partir de el nombre de un archivo de texto, generar la codificación con compresión de huffman los caracteres junto a la estadística de su frecuencia.

Análisis del problema:

Dado un texto se busca encontrar una manera de guardarlo ocupando menos espacio. Primero es necesario darse cuenta que un archivo dentro de si tiene información y dicha información tiene una codificación en bits especifica y es esto último lo que utilizan el espacio en la memoria. Entonces para poder solucionar ese problema es natural buscar dentro del archivo mismo algo que nos permite ocupar menos espacio. La estrategia tomada por Huffman fue, dándose cuenta de que cada pedazo de información tiene una codificación en bits, darle un nuevo valor a la codificación en bits de los caracteres de acuerdo a que tanto se repite esa letra en el texto.

La idea es brillante pues se evitan muchas redundancias (si es que existen tendencias) pues ahora un valor que era representado por 16 bits puede ser reducido a 4 o incluso 2 bits con esta codificación dependiendo del caso. De esa manera ahora el problema de optimizar la memoria queda resuelto pero el aún queda sujeto a la frecuencia de los pedazos de información en el archivo.

Solución del problema:

La solución del problema es claro, para optimizar la utilización de memoria de un archivo hay que implementar la codificación de Huffman. Para este caso particular se busca lograr que un archivo de texto sea codificado, es decir que las letras que componen el texto sean codificadas.

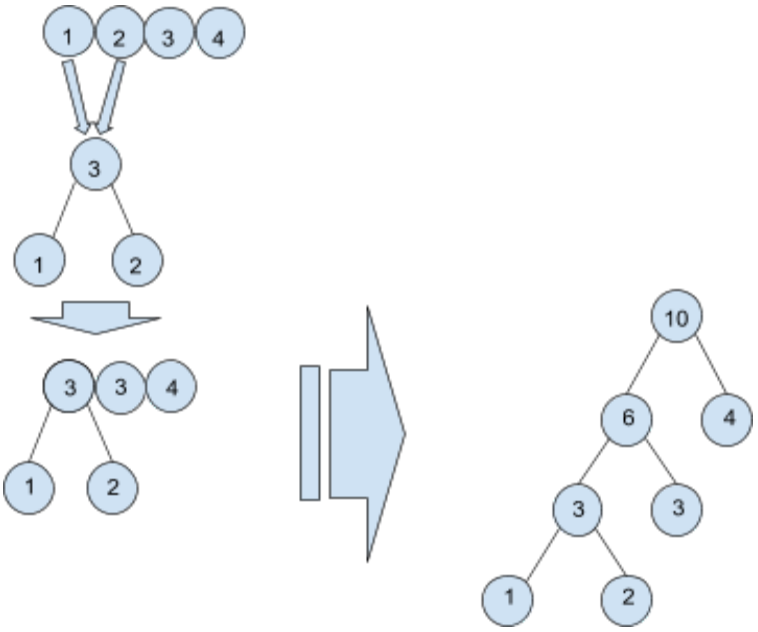
Para esto primero es necesario realizar un análisis estadístico del archivo de texto, hay que encontrar la frecuencia de cada carácter. Para lograr lo anterior simplemente se realizo una lectura del texto entregado por el usuario y fue guardado en un string. Una vez recuperado se creó un arreglo donde se guardaron cada uno de los caracteres con sus frecuencia respectiva.

Luego se construyó un heap a partir de los nodos con la restricción de orden “frecuencia del padre siempre es menor que la del hijo”. Para poder crear este heap primero se dispuso un arreglo llenos de nodos “nulos” (los cuales son simplemente un nodo comparable a los anteriores para marcar el fin del heap en el arreglo). Después se creó el método insertar, que logra que se preserve la restricción de orden cuando se agrega un elemento nuevo al heap. Así se fue insertando ordenadamente de a uno cada uno de los nodos con el par caracter frecuencia.

Luego, tras construir el heap, se empieza a implementar la compresión de Huffman. El algoritmo de la compresión consiste en tener un arreglo, extraer los dos nodos menores de este arreglo, crear un nuevo nodo nulo que contenga la suma de las frecuencias de los dos nodos anteriores y tenga como hijos a estos, y finalmente la inserción de este último nodo en el heap. Este proceso se repite hasta que solo quede un elemento en el arreglo que es un árbol binario con los caracteres en las hojas y la suma de las frecuencias de todos los caracteres del texto en la raíz.

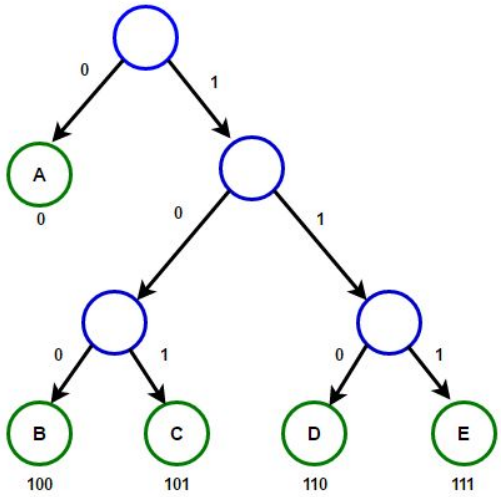
```
while(a[1].frec != Integer.MAX_VALUE || a[1].l != '\u0000'
|| a[1].der != null){
      Arbol H1 = a[0];  //se saca del heap al hijo derecho
      a = extraerMin(a);
      Arbol H2 = a[0]; //se saca del heap al hijo izquierdo
      a = extraerMin(a);
      Arbol P = new Arbol('\u0000', H1.frec + H2.frec, H1,
H2);
       a = insert(a, P);
      }
```

Aquí hay un ejemplo visual de cómo construir un árbol de la compresión de un arreglo de 4 elementos para ilustrar el procedimiento anterior:



Para poder hacer lo anterior se debió crear el método extraerElMinimo, que extrae el mínimo del heap (la raíz), trayendo a la raíz al último elemento del heap y rotando con sus hijos para mantener la restricción del árbol.

Finalmente para obtener la codificación de cada uno de los caracter basta seguir la ruta desde el nodo raíz hasta el carácter agregando 0 a el string de la codificación cada vez que se bajó por la izquierda y un 1 cada vez que se bajó por la derecha y se obtendrá la codificación del carácter.



Modo de uso:

Para la utilización del código es importante tener en cuenta que los nombres de los archivos ingresados en la consola deben terminar con su extensión .txt de lo contrario se tendrá un error. Esto quiere decir que si es que quiero comprimir el archivo “Urfaust” es necesario ingresar en la consola como archivo de entrada *Urfaust.txt* y si es que quiero que mi archivo de salida se llame “ComUrfaust” debo ingresar *ComUrfaust.txt*

También es necesario que se modifique (dentro del código, línea 51) la ruta de donde esta el archivo de texto a comprimir. En esta línea:

```
String texto = leerTxt("C:\\Users\\Bacchanalia\\Desktop\\T5\\" + entrada);
```

hay que sustituir *C:**Users**Bacchanalia**Desktop**T5*\\ por la ruta del archivo (sin incluir el nombre del archivo) y utilizando siempre doble backslash para evitar ciertos errores. Es decir si la ruta del texto es *C:**Users**Usuario**Desktop**Urfaust.txt* se debe reemplazar en el código lo anterior por *C:**Users**Usuario**Desktop*\\.

Resultados:

Como resultado de la implementación se obtuvieron archivos de textos en los que se explicita cada carácter, su codificación de huffman y su frecuencia relativa (en porcentaje). Como ejemplo se utilizó el texto *ElCidC1.txt* llegando al siguiente resultado para algunas letras:

```
Carácter Cód ASCII(dec.) codificación frecuencia
' ' 32 100 16.557707%
'a' 97 110 10.595872%
'e' 101 1000 8.990342%
'o' 111 1010 6.8636785%
's' 115 1011 5.8775806%
'r' 114 1101 5.1146026%
'n' 110 1110 4.9866595%
'l' 108 1111 4.7323337%
'd' 100 10001 4.1643915%
'i' 105 10100 3.3483639%
'u' 117 10110 2.920847%
't' 116 11010 2.6634004%

.
.
.
```

Una vez obtenida la codificación de cada una de las letras se pasó a calcular cuánto era el largo del texto comprimido (largo codificación*frecuencia) y a compararlo lo con el largo del texto original (bytes*8).

Nombre del Archivo	Largo del Archivo original	Largo del archivo comprimido
ElCidC1.txt	524584	288434
Hamlet.txt	1515024	857253
Urfaust.txt	2560944	1474779

Como se esperaba, la cantidad de bits de la compresión es menor a la cantidad de bits del texto original y algo como un 40% menos de hecho.

Teniendo en cuenta que el problema era el de optimizar el espacio ocupado en la memoria se puede concluir a partir de los resultados que la compresión de Huffman es efectiva a la hora de hacerlo sin embargo tiene la limitación de que funciona a partir de información que quizá no siempre se tiene (frecuencia de aparición de un pedazo de información.